Object-oriented Python

This lab will take you through the basics of object-oriented programming in Python: defining and using a class, its properties and its member functions.

20.1 Walkthrough: A Person class

Create a separate Python file called `person.py`:

```python
class Person(object):
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname
```

Remember, that the `__init__` function name has two underscores before and after the `init` and that (for now at least) every method you define in a class should have the first argument called `self`.

When you run this module now in IDLE (by pressing F5) you will have access to the `Person` class and you can start creating `Person` objects:

```python
>>> p = Person('James', 'Curran')
>>> p
<__main__.Person instance at 0x74918>
```

Notice that Python uses angle brackets with the name of the class and the object’s address in memory to represent a user defined class by default. You can now also access and modify the properties of the `Person` object:

```python
>>> p.fname
'James'
>>> p.lname
'Curran'
>>> p.fname = 'Bob'
>>> p.fname
'Bob'
```

Alternatively, you can run Python in Unix with the `-i` option which runs the `person.py` file and then starts the interactive interpreter:

```bash
% python -i person.py
>>> p = Person('James', 'Curran')
...```

You should also experiment with creating multiple objects so that you can see they are independent of each other:

```python
>>> p1 = Person('James', 'Curran')
>>> p1
<__main__.Person instance at 0x74918>
>>> p2 = Person('Tara', 'Murphy')
>>> p2
<__main__.Person instance at 0x748c8>
```

You can tell they are separate objects because they are stored at different addresses (here `0x74918` for James and `0x748c8` for Tara).
The next thing to do is to add some functionality to the `Person` class. Firstly, we’ll write a **method** to return person’s full name. A method is just a function that operates on an object. The `self` argument that all methods need is the object that the function will operate on.

```python
class Person(object):
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname

    def name(self):
        return self.fname + ' ' + self.lname
```

The `def` statement needs to be indented to indicate that the method is part of the class definition (which is also true for the `__init__` method. NB: the `__init__` method has the special name **constructor** since it is responsible for creating (or constructing) the object.

We can now create a `Person` and call the `name` method:

```python
>>> p = Person('James', 'Curran')
>>> p.name()
'James Curran'
```

When we update a property of the object, methods using that property will return updated results:

```python
>>> p.lname = 'Bond'
>>> p.name()
'James Bond'
```

As mentioned in lectures, you may not want programmers who are using your class to have access to their internal representation. That’s because you really want the internal state of your objects to remain consistent (i.e. have sensible values at all times). To see how easy it is to break things, try:

```python
>>> p.lname = 123
>>> p.name()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "person.py", line 7, in name
    return self.fname + ' ' + self.lname
  TypeError: cannot concatenate 'str' and 'int' objects

```

This looks like an error inside the `Person` class but it was caused by a programmer incorrectly setting the last name to a value that doesn’t make sense.

The way to avoid this is to create **getter** and **setter** methods that get and set the individual properties:

```python
class Person(object):
    def __init__(self, fname, lname):
        self._fname = fname
        self._lname = lname

    def fname(self):
        return self._fname

    def set_fname(self, fname):
        self._fname = fname

    def lname(self):
        return self._lname

    def set_lname(self, lname):
        self._lname = lname

    def name(self):
        return self._fname + ' ' + self._lname
```

James Curran and Tara Murphy
We’ve also changed the properties so that they have an underscore in front of the name _fname and _lname. This indicates that they are private properties and should not be directly accessed. However, Python, unlike many other OO languages does not enforce this, you can still access them, but it is at your own peril!

So to use the new version of Person you always call methods:

```python
>>> p = Person('James', 'Curran')
>>> p.fname()
'James'
>>> p.set_fname('Bob')
>>> p.name()
'Bob Curran'
```

Now we can add checks in our setter functions that make sure the values passed in make sense. For example, we can make it illegal to set a blank first name, using something like this:

```python
class Person(object):
    ...
    def set_fname(self, fname):
        fname = fname.strip()
        if not fname:
            raise ValueError('the first name cannot be blank')
        self._fname = fname
    ...
```

This method also does the work of removing any whitespace around the first name before setting the property.

### 20.2 Other special methods

So far, you’ve seen the _init_ method, which Python calls when it creates new objects (like Person above). You can also define many other special methods, that give your objects special powers!

The simplest method is _str_ which should return a string representation of your object:

```python
class Person(object):
    ...
    def __str__(self):
        return self._fname + ' ' + self._lname
```

The `__str__` method gets called whenever Python tries to convert your object into a string, e.g. in a `print` statement, or explicitly with `str`:

```python
>>> import person
>>> p = person.Person('James', 'Curran')
>>> p
<Person.Person instance at 0x24f558>
>>> str(p)
'James Curran'
```

Another method is _repr_ gets called whenever Python tries to display an object in the Python interpreter:

```python
class Person(object):
    ...
    def __repr__(self):
        return '<Person %s %s>' % (self._fname, self._lname)
```

And here is the `__repr__` method in action:

```python
>>> p
<Person James Curran>
```

The complete list of special methods is available here:

http://docs.python.org/reference/datamodel.html#basic-customization
20.3 Exercises

20.3.1 The Ball class

In this exercise, you’ll go back through the example we did in lectures:

1. Create a class that represents a ball.
2. It should have properties for colour and radius.
3. Write a method to calculate the diameter, circumference and volume. You’ll need to import the \texttt{math} module to get access to \texttt{math.pi} for the value of \pi.
4. Try creating ball objects and calculating their circumference and volume.
5. Now change your implementation to use getters and setters, hiding direct access to the properties. Add a setter for the diameter as well that
6. Make the \texttt{set\_radius} and \texttt{set\_diameter} functions raise a \texttt{ValueError} exception if the value is less than or equal to zero.
7. Now, change the internal implementation so that it stores the diameter not the radius.

Notice that with a well defined interface other programmers can be completely unaware of how the internal representation works. We can change from storing the radius to storing the diameter, but you don’t need to change anything that uses the Ball class.

20.3.2 The Particle class

Create a class that stores the properties of a particle for a Physics simulation. It should store the current position of the particle in two dimensions ($x$ and $y$) and the velocity of the particles ($dx$ and $dy$).

The particle class should also provide a \texttt{move} method which updates the position on the basis of the velocity. Every time you call \texttt{move} it should add the velocity to the current position (i.e. each call to \texttt{move} corresponds to one second and the velocity is in metres per second).

Next, change the \texttt{move} method so that it keeps the particles bounded inside a box.

Finally, add some special methods so that you can:

1. print out the particle details (\texttt{\_str\_} method)
2. display the particle details in the interpreter (\texttt{\_repr\_} method)
3. add and subtract particles from each other (\texttt{\_add\_} and \texttt{\_iadd\_} methods etc)
4. scale a point by a scalar value (\texttt{\_mul\_} and \texttt{\_imul\_} methods)

20.3.3 The Dict class

Here’s a real challenge: implement your own Python dictionary class using the builtin \texttt{hash} function. Using the special methods, you can make it work exactly like a normal dictionary!